# Problem 1. Sino The Walker

Sino is one of those people that lives in SoftUni. He leaves every now and then, but when he leaves he always takes a different route, so he needs to know how much time it will take for him to go home. Your job is to help him with the calculations.

You will receive the **time that Sino leaves** SoftUni, the **steps taken** and **time for each step in seconds**.

You need to print the exact **time** that he will **arrive at home in the format specified**.

## Input / Constrains

* The first line will be the time Sino leaves SoftUni in the following format: **HH:mm:ss**
* The second line will contain the **number of steps** that he needs to walk to his home. This number will be an integer in range **[0…2,147,483,647]**
* On the final line you will receive the **time in seconds** for each **step**. This number will be an integer in range **[0…2,147,483,647]**

## Output

* Print the **time of arrival** at home in the following format:
  + **Time Arrival: {hours}:{minutes}:{seconds}**
* If hours, minutes or seconds are a single digit number, add a zero in front.
* If, for example, hours are equal to zero, print a **00** in their place. The same is true for minutes or seconds.
* Time of day starts at **00:00:00** and ends at **23:59:59**

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 12:30:30  90  1 | Time Arrival: 12:32:00 |
| **Input** | **Output** |
| 23:49:13  5424  2 | Time Arrival: 02:50:01 |

# Problem 2. SoftUni Karaoke

SoftUni cultivates talent whether it's coding talent or something else and in this case, something else is singing. Since you love music you want to take part in the event but as a programmer you simply lack the "something else" so your job is to make the software to track participants' awards.

**On the first line**, you will receive a **list with all participants that applied for performance**.

**On the second line**, you will receive **the list with all available songs**.

**On the next lines**, until the "**dawn**" **command**, you will get the **names of people**, **the song that they are performing** on stage and **the award they get** from the audience.

However, not every time the plan goes according to schedule. In other words, **everyone** (listed or not) **can go on stage and perform a song that is not even available** and he **still gets awards from the audience**. However, you should **record only the awards for listed participants** that **perform songs available in the list**. In case someone is **not listed** or sings a **song that is not available** you **should not record neither the participant, nor his award**.

When the "dawn" comes, you need to **print all participants**, the **count of the unique awards they received** and all unique awards. Participants should be **sorted by number of awards** in **descending order** and **then by participant name alphabetically**. Awards should be **sorted in alphabetical order**.

### Input

* On the first line, you will receive list with all participants that applied for performance in the format: **"{participant}, {participant} … {participant}"**
* On the second line, you will get all available songs in the in the format: **"{song}, {song} … {song}"**
* On the next lines, until the **"dawn"** command you will receive every stage performance in the format: **"{participant}, {song}, {award}"**
* Performances and songs will be **separated by a comma and a single or multiple white spaces**

## Output

* Print all participants along with the number of unique awards they won in the format:
  + **"{participant}: {award count} awards"**
  + **"--{award}"**
* Print participants **sorted by unique awards count in descending order**. If two participants have the same unique award count**, sort them alphabetically by name**
* Print unique **awards** for every participant **sorted alphabetically**
* If there are **no awards,** print **"No awards"**

## Constrains

* The number of total participants will be in range [1 … 100]
* The number of total songs will be in range [1 … 100]
* The input will always end with the "dawn" command

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Trifon, Vankata, Gesha  Dragana - Kukavice, Bon Jovi - It's my life, Lorde - Royals  Gesha, Bon Jovi - It's my life, Best Rock  Vankata, Dragana - Kukavice, Best Srabsko  Vankata, Dragana - Kukavice, Best Srabsko  Vankata, Dragana - Kukavice, Stiga Tolko Srabsko  Vankata, PHP Web, Educational 101  dawn | Vankata: 2 awards  --Best Srabsko  --Stiga Tolko Srabsko  Gesha: 1 awards  --Best Rock |
| Gesha  Bon Jovi - It's my life  Gesha, Bon Jovi - It's my life, Best Rock  Vankata, Dragana - Kukavice, Best Srabsko  Vankata, Dragana - Kukavice, Stiga Tolko Srabsko  Vankata, PHP Web, Educational 101  dawn | Gesha: 1 awards  --Best Rock |
| Sino  Vasko Naidenov - Nova Godina  dawn | No awards |

**Problem 3 – Endurance Rally**

The goal of the Endurance Rally is, simply, to finish the race.

You are given **the names of the participants**, **the track layout** and **the** **checkpoint** indexes.

The starting fuel of each participant is **equal to the ASCII code** of the **first character** of his name.

Track layout consists of zones represented by numbers, given on a single line separated by a single space. Every number **represents** the **fuel given** or the **fuel taken** by the current zone of the track:

* If the current zone **is a checkpoint**, the value of the zone is **added** to the driver's **fuel**.
* If the current zone **is not a checkpoint**, the value of the zone is **subtracted** from the driver's fuel.

Zones are **indexed**. Indexes are sequential and always **start from zero** (like an array).

The **checkpoints** are numbers, representing indexes that show if **a zone of the track** **is a checkpoint**. For example, you are given checkpoints 0, 3 and 5, that means that zones at index 0, 3 and 5 of the track are checkpoints and therefore provide fuel for the driver.

Given this information, you need to **check if a driver can finish** and **print the fuel that he is left with**. If a driver **can't finish** you need to **print the zone he managed to reach**.

**Input**

The input will be read from the console. The input consists of **exactly three lines**:

* The first line holds the drivers' names separated by a space: **"{driver 1} {driver 2} … {driver N}"**
* On the second line is the track layout (zones) in the form of numbers separated by a space: **"{zone 0} {zone 1} … {zone N}"**
* On the third line are the checkpoint indexes also separated by a space: **"{index 0} {index 1} … {index N}"**

**Output**

Print all drivers **in the order of their appearance**, each on a separate line:

* If the driver finished, print his name and fuel left to the second digit after the decimal point in the format: **"{driver name} - fuel left {fuel points}"**
* If the driver could not finish, print: **"{driver name} - reached {zone index}"**

**Constrains**

* Drivers count will be in the range [0 … 200]
* Zone fuel will be a floating-point number
* Checkpoints will be integers in the range [-231 … 231 - 1]

**Examples**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| Garry Clark  69 **1** **15** 5  1 2 | Garry - fuel left 13.00  Clark - reached 0 | Zones 1 and 2 -> checkpoints.  Garry ('G' = 71)  -> 71 - 69 + 1 + 15 - 5 = 13.00  Garry finished with 13 fuel  Clark ('C' = 67)  -> 67 - 69 = -2  Clark reached 0 |
| Garry Clark Larry  **4** 5 12 **0** 8 **7** 13 22 **5.5** 26  0 3 5 8 | Garry - fuel left 1.50  Clark - reached 9  Larry - fuel left 6.50 |  |
| Garry  -29 **-5.0** **-5.0**  1 2 | Garry - fuel left 90.00 |  |

**Problem 4 – Winning Ticket**

Lottery is exciting. What is not, is checking a million tickets for winnings only by hand. So, you are given the task to create a program which automatically checks if a ticket is a winner.

You are given a **collection of tickets separated by commas and spaces**. You need to check every one of them if it has a winning combination of symbols.

**A valid ticket should have exactly 20 characters**. The winning symbols are '**@**', '**#**', '**$**' and '**^**'. But in order for a ticket to be a winner the symbol should uninterruptedly repeat for at least **6 times** in both the **tickets left half** and the **tickets right half**.

For example, a valid winning ticket should be something like this:

"**Cash$$$$$$Ca$$$$$$sh**"

The left half "**Cash$$$$$$**" contains "**$$$$$$**", which is also contained in the tickets right half "**Ca$$$$$$sh**". A winning ticket should contain symbols repeating up to 10 times in both halves, which is considered a Jackpot (for example: "**$$$$$$$$$$$$$$$$$$$$**").

**Input**

The input will be read from the console. The input consists of a **single line** containing all tickets **separated by commas and one or more white spaces** in the format:

* **"{ticket}, {ticket}, … {ticket}"**

**Output**

Print the result for every ticket in the order of their appearance, each on a separate line in the format:

* **Invalid ticket - "invalid ticket"**
* **No match - "ticket "{ticket}" - no match"**
* **Match with length 6 to 9 - "ticket "{ticket}" - {match length}{match symbol}"**
* **Match with length 10 - "ticket "{ticket}" - {match length}{match symbol} Jackpot!"**

**Constrains**

* Number of tickets will be in range [0 … 100]

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cash$$$$$$Ca$$$$$$sh | ticket "Cash$$$$$$Ca$$$$$$sh" - 6$ |
| $$$$$$$$$$$$$$$$$$$$ , aabb , th@@@@@@eemo@@@@@@ey | ticket "$$$$$$$$$$$$$$$$$$$$" - 10$ Jackpot!  invalid ticket  ticket "th@@@@@@eemo@@@@@@ey" - 6@ |
| validticketnomatch:( | ticket "validticketnomatch:(" - no match |